REAL-TIME DETECTION OF FOREGROUND IN VIDEO SURVEILLANCE CAMERAS USING CUDA

A Thesis

by

LAKSHMI VENUGOPAL

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Tie Liu
Committee Members,	Xiaoning Qian
	Jean-Francois Chamberland-Tremblay
	Anxiao Jiang
Head of Department,	Miroslav Begovic

May 2018

Major Subject: Electrical Engineering

ABSTRACT

The rapid growth of video processing techniques has led to remarkable contributions in several applications such as compression, filtering, segmentation and object tracking. A fundamental task of video surveillance cameras is to detect and capture major moving objects (foreground).

Processing video frame by frame is complex and difficult for real time applications. GPUs have led to significant advancements in the field of image/video processing especially in real time applications. In this work, we make use of the parallel computing capacity of GPUs to speed up the runtime of foreground detection algorithm.

The focus of the thesis is to accelerate the runtime of the algorithm by parallelizing the time consuming portions. The final goal would then be to analyze and come up with the optimal parallelization technique(s) that give(s) the best performance.

ACKNOWLEDGMENTS

I would like to thank Texas A&M University for giving me an opportunity to present my thesis work. I express my sincere gratitude towards my advisor Dr. Tie Liu for helping me in completing the thesis. Without his constant support and guidance this thesis would not have been successful. Also I would like to thank other committee members who have supported me in my defense. I would like to thank my family who helped me in providing an atmosphere to complete my thesis. I would like to specially thank my brother for providing constant inputs and feedbacks in my thesis. I would like to thank all my friends and colleagues at Texas A&M University for their moral support. I thank God Almighty for helping me finish my thesis work.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supervised by a thesis committee consisting of Dr. Tie Liu, Dr. Xiaoning Qian and Dr. Jean-Francois Chamberland-Tremblay [advisor - Dr. Tie Liu] of Electrical Engineering Department and Dr. Anxiao Jiang of Computer Science and Engineering department.

The algorithm described in Chapter 2 was studied by Han Guo, Chenlu Qiu and Namrata Vaswani of Department of Electrical and Computer Engineering, Iowa State University and were published in 2014 in IEEE Transactions on Signal Processing

All other work conducted for the thesis was completed by the student independently.

Funding Sources

Graduate study was supported by Texas A&M University.

NOMENCLATURE

CPU	Central Processing Unit
GPU	Graphics Processing Unit
BLAS	Basic Linear Algebra Subroutines
GEMM	General Matrix-matrix multiplication
CUDA	Compute Unified Device Architecture
OpenCV	Open Source Computer Vision
PracReProCS	Practical Recursive Projected Compressive Sensing
RTSP	Real Time Streaming Protocol
IP	Internet Protocol
fps	frames per second
PCIe	Peripheral Component Interconnect Express
SVD	Singular Value Decomposition
PCA	Principal Component Analysis
HDMI	High-Definition Multimedia Interface
USB	Universal Serial Bus
MPI	Message Passing Interface
OpenMP	Open Multi-Processing

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

1.1 Modern challenges faced by digital media

As a consequence of rapid growth of digitalization, the current generation digital media poses formidable challenges in storage, processing and access. With the enormous competition among digital providers and rapidly advancing research, the digital media has shown significant growth in various domains. In today′s world, the visual media has surpassed any others in the field of media entertainment. The integration of digital media along with the internet has enabled the world to connect with each other more feasibly. However, massive real-time transfer and processing of extremely-dense videos become technically challenging even with generous amounts of computational resources.

The challenges that evolved with the digital media revolution has led to research in the areas of image and video processing. While most of the video processing techniques like filtering, color enhancement, noise reduction are already in practice, advancements are still going in the field of compressive sensing [1] for applications like face recognition, magnetic resonance imaging and photography.

1.2 PracReproCS algorithm overcoming a major challenge in foreground detection

Modern video surveillance cameras are equipped with video processing features like auto-focus adjustment, motion sensors, internal hard drive and infrared lights, the most important one being the detection of moving target. Currently, most video cameras are able to capture moving objects with reasonable accuracy, but the challenge lies in identifying the moving target even in the presence of background noise (such as moving tree or playing television).

One of the latest algorithm developments in the detection of a moving target is the Practical Recursive Projected Compressive Sensing algorithm (Prac-ReProCS) of Guo et.

al. [2]. The algorithm has shown promising results even in the presence of background noise (such as a moving tree or a playing television). However, the parent program is coded to work on a single core CPU and is compatible only with Matlab [3]. This restricts its integration with real-world high resolution video cameras. The implementation then becomes computationally intensive and ends up consuming generous amounts of computational resources. In this work, we try to enhance the performance of the Pra-ReProCS algorithm through parallel computing. One particular application is to provide optimal lighting only to the area of interest (foreground). Parallelization of intensive computations in the foreground detection procedures of the algorithm will help in real time lighting optimization.

1.3 CPU Vs GPU

In the earlier days, speedup in single-core CPUs was achieved by increasing the clock rates and by providing instruction-level parallelism. Eventually, multi-core CPUs were designed for running processes simultaneously in different cores, thereby utilizing the full hardware to achieve much scalable speedups. Thread level parallelization like OpenMP is much faster than distributed memory parallelization approach like MPI [4]. This is because that thread level parallelism is performed on a shared memory architecture that results in lesser time for data communication between processes. However, thread level parallelism in CPU based programs are limited by the number of threads available. On the other hand, a GPU based supercomputer outperforms CPU architecture for highly parallelizable selective problems as they have larger number of threads for parallel computing. However, explicit communication (but less frequent, depending on the problem of interest) is required when tranferring data between GPU and CPU in case of GPU parallelization.

CPU parallelization is better when it comes to performing different tasks at the same time whereas GPU parallelization is superior when same task needs to be performed in all

the pixels of an image or in all the elements of the matrix. In the susequent sections, we will see that the kind of problem we are interested is worth trying to parallelize on a GPU platform.

1.4 Problem statement

Recently, GPUs have gained attention in accelerating the runtime through its parallel architecture. GPUs are used in several applications like computers, mobile phones, and gaming consoles that involve intensive computations. GPUs have proven to obtain faster results than CPUs for several time consuming algorithms by parallelizing large blocks of code [5],[6].

The focus of the thesis is to accelerate the runtime of the Prac-ReProCS algorithm by parallelizing the time consuming portions of the parent serial program. In order to enable multiple kinds of parallelization techniques such as OpenMP, MPI or GPU-based CUDA [7], our first objective is to implement the Prac-ReProCS algorithm in C++. We also make use of pre-written OpenCV [8] library functions that are easily parallelizable. The next step is to detect and parallelize the time consuming parts of the code by launching CUDA kernel and dividing the task among different GPU threads. The final goal would then be to analyze and come up with the optimal parallelization technique(s) that give(s) the best performance.

In this work, we focus on two major objectives: (a) implementation in C++ and (b) parallelization of the code on a GPU platform. Jetson TK1 development board is used to parallelize PracReProCS algorithm using CUDA C++. CUDA is a parallel programming platform developed by NVIDIA to facilitate the use of GPUs. Its compatibility to work with C++ makes it easier to parallelize the code. The board specifications are discussed in the next section. The PracReProCS algorithm is briefly discussed in the next chapter.

1.5 Jetson TK1 development board

In this work, we exploit the computing power of GPUs using NVIDIAs Jetson TK1 development board [9]. The board has Kepler GPU with 192 CUDA cores and 4-Plus-1 quad-core ARM Cortex-A15 CPU. The board was chosen to perform different parallelization techniques, the major one being the CUDA parallel programming. OpenMP is an API that helps in parallelization of multiple cores in CPU. Therefore, the four CPU cores in this board provide more resources for CPU parallelization techniques like OpenMP and MPI. This further improves the efficiency without the difficulty of copying data to and from device memory.

The OS was flashed into the Jeston board using Jetpack software development kit [10]. The Jetpack installer required Ubuntu 14.04 in the host device for flashing the OS. The important features in NVIDIA Jetpack SDK like Cuda 8.0, Linux for Tegra driver package (32bit Ubuntu 14.04), OpenCV library [8] makes it more easier to improve the efficiency of the algorithm using GPUs.

2. PRACREPROCS ALGORITHM

PracReProCS algorithm is developed by Guo.et.al [2] and is intented to capture a moving target in a video. The difference of this algorithm from other algorithms in this field [11],[12] lies in the fact that PracReProCS algorithm considers the presence of background noise, which makes the real target detection in the foreground more meaningful. The algorithm has demonstrated the detection of a moving object (foreground) even in the presence of slowly moving background or small disturbances in the background.

Here, the foreground is represented as sparse vectors (S) and the background is considered as dense vectors (L). Given a measurement vector, M, the ultimate goal of the algorithm is to separate the sparse signal from M, where M = S + L.

2.1 Training

Let I_t be the image at time t, F_t and B_t be the foreground (moving target) and the background at time t respectively. During training stage, since the video only contains background frames, the mean of the training set is represented as μ. Let,

$$L_t = B_t - \mu, M_t = I_t - \mu \tag{2.1}$$

Measurement vector at time t is given by $M_t = S_t + L_t$. For training period, i.e, for t = 1 to t_{train}, $M_{train} = L_t$. This means that a set of background only frames are fed as the training data, thereby obtaining an initial estimate of subspace of L_t. For testing purposes, the data from the previous frame is used to get the information of the current frame, i.e, L_t is found using L_{t-1} and S_t using S_{t-1}.

Let $T_t = supp(S_t)$ represent the support set of S_t, i.e, indices where S_t is non-zero. An approximate initial basis for background denoted as, $\hat{P}_0$ is same as initial basis for M_{train}

as only background frames constitutes M_{train}.

$$\hat{P_0} = approx - basis(M_{train}, b\%) \tag{2.2}$$

The initial basis is found by computing singular value decomposition of M ($M \overset{\text{SVD}}{=} U\Sigma V'$) and considering only $b\%$ of the left singular values (U). b represents the percentage of how low rank the matrix is.

2.2 Testing

Training is mainly done to calculate the initial basis of L_t. After getting initial estimate for basis, we start the testing phase and the remaining steps are calculated for each testing frame. Firstly we find the space orthogonal to P_{t-1} at time t, which is given by,

$$\phi_t = I - \hat{P}_{t-1} * \hat{P}_{t-1} \tag{2.3}$$

Projected measurement vector, y_t is obtained by projecting M_t onto ϕ_t.

$$y_t = \phi_t * M_t = \phi_t * (S_t + L_t) \tag{2.4}$$

$$y_t = \phi_t * S_t + \beta_t, where \beta_t = \phi_t * L_t \tag{2.5}$$

Since the background is slowly moving and we do not have any information of L_t, we approximate L_t to L_{t-1}. ϕ_t and L_{t-1} are almost perpandicular and projecting ϕ_t to L_{t-1} is approximately 0 as most of the components nullify. Therefore, β_t can be considered as a small noise.

Sparse vector, S_t is recovered from projection y_t, using l1 minimizaton.

$$min_x||x||_1 s.t. ||y_t - \phi_t * x||_2 <= \zeta \tag{2.6}$$

Here the noise term is ζ and we assign the value of ζ as $||\beta_t||_2 = ||\phi_t * L_{t-1}||_2$. Once we find S_t, we calculate L_t as $\hat{L}_t = M_t - \hat{S}_t$.

2.3 Background subspace updation

The final step is to update the subspace of background P_t. We need to update P_t only if there is a change in the subspace. To detect the change in subspace, we set a noise threshold, $\hat{\sigma}_{min}$ which is the $\hat{r}^{th}$ largest singular value of the training dataset. Let us represent the basis matrix for the previously computed subspace as $\hat{P}_{j-1}$. We find the projection of the last $\alpha \hat{L}_t s$ frames perpendicular to $\hat{P}_{j-1}$ for every α frames. Then, we calculate the SVD of the projection. The subspace has changed if there are any singular values above $\hat{\sigma}_{min}$ in the SVD computation. If there is a change in subspace, we calculate Projection PCA (p-PCA). This is done by estimating $c_{j,new}$ top singular values of the projection that are greater than $\hat{\sigma}_{min}$. The new projected matrix, $\hat{P}_{(j),new}$ is then appended to the previous basis, $\hat{P}_{j-1}$ to get the final basis $\hat{P}_t$.

$$\hat{P}_t = [\hat{P}_{j-1}, \hat{P}_{(j),new}] \tag{2.7}$$

Thus the final basis, $\hat{P}_t$ will contain the updated information (new directions) of the backgound if the background has slightly changed.

3. REAL-TIME PRACREPROCS

Being able to execute PracReProCS algorithm in real-time will benefit a multitude of applications in the field of image processing. In this work, the aim is to develop and study the benefits of a real-time PracReProCS that is integrated with a survillience camera equipped with variable LED lighting technology. This chapter describes the steps that were performed to develop a PracReProCS that can work in realtime with low latency.

3.1 Configuring the board and preprocessing

The PracReProCS algorithm was tested in videos having a resolution of 100X100. Three hundred frames were used to train the dataset, while testing was performed on a database with 200 frames. The datasets contain the grayscale pixel intensity values of the frames. The computational time of the whole algorithm was about 80 seconds. The distribution of the computational time in main parts of the code is shown in Table 3.1. After further profiling of the code, it was observed that the testing took approximately half a second per frame. Though this seems to be a reasonably good execution speed, in real time scenario (where there are 15 or more frames per second), half a second becomes a significant amout of delay. In addition, the algorithm takes much more time for high resolution videos.

Part of code	Execution time
Training	0.4 seconds
Testing	10 seconds
Miscellanious part and writing video to file	70

Table 3.1: Distribution of computational time

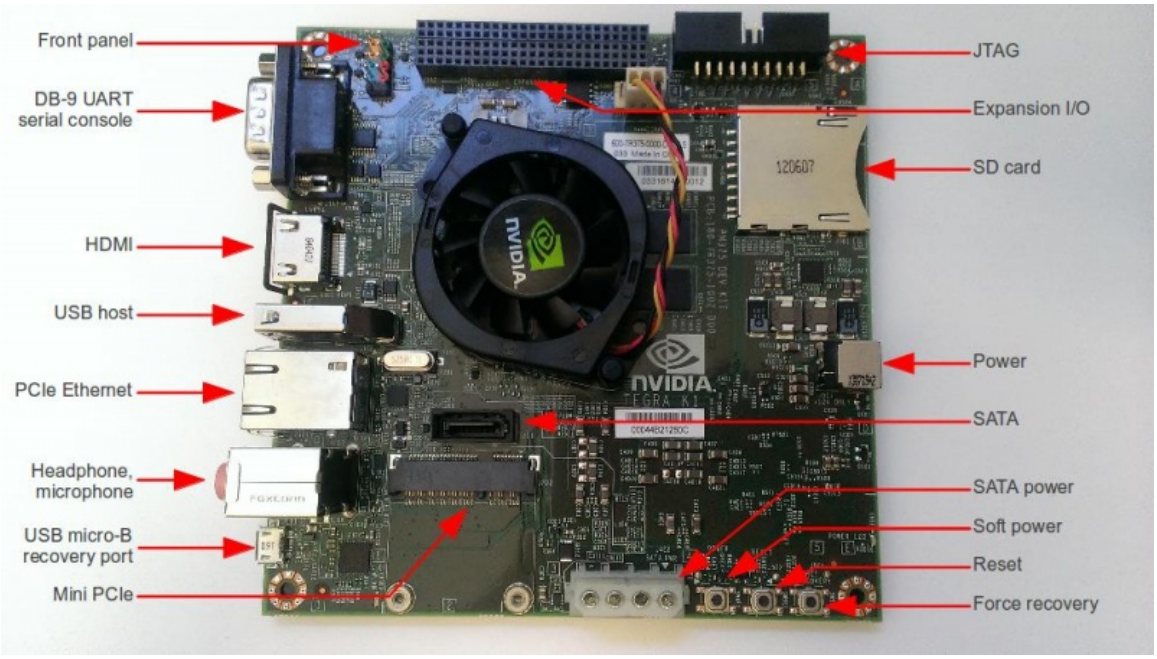

Figure 3.1: Jetson TK1 development board

To study the real-time performance of the PracReProCS algorithm, we experiment the code on a video surveillance IP camera. The IP camera has a frame rate of 15 fps and has a resolution of 1920x1080. It is connected to the PCIe Ethernet port (shown in Figure 3.1) of Jetson TK1 development board to feed the video input. Due to memory constraints of the development board [13], each frame of the video is downsized by a factor of 8. Details of the Jetson TK1 board is briefly discussed in section 1.5. Further the monitor is connected to the HDMI port and the keyboard and mouse are connected to the USB port via USB hub. After connecting the adapter to power, the soft power button is pressed to switch on Jetson TK1.

The IPV4 settings of the computer is then changed to the settings of the IP camera (IP address, netmask, gateway, DNS servers). The live video is streamed using RTSP protocol by accessing the IP address and port number of the camera as shown in the above figure.

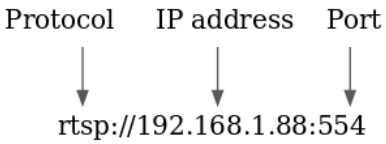

Figure 3.2: Streaming video using RTSP

3.2 Determining training and testing data

The training dataset for the origianal algorithm of Guo.et.al [2] was considered as background only frames. For all the videos that were tested, the training dataset was readily available. But for practical cases, it is difficult to go with this assumption as we do not have perfect information about the background subspace in most cases. Therefore, determining the relevant training data is an important study since the accuracy of the algorithm is highly dependent on the characteristics of the training dataset.

The initial thought was to train the video periodically. For instance, the training frames are taken from camera for 20 seconds and the next available frames are tested for 5 minutes. This approach gives high accuracy as training repeatedly would account for major changes in the background within the next training period. But when we consider a case where the foreground is present during the training, the algorithm gives significant errors during testing. This is because the algorithm assumes that the training contains only background frames.

One solution to the above mentioned problem would be to give a sufficiently large training dataset. This ensures that the training data contains all possibe background information. The computational time for finding the initial basis of the background increases with an increase in the number of training datapoints. But we do not need to take the training time into account mainly due to two reasons. Firstly, the training needs to be done only once, after installing the camera. Secondly, since the initial basis for background contains

most of the background information, small changes in the background gets updated to the basis in the testing period.

Another solution to the problem would be to start the training whenever the foreground is not present. In this case, we find the largest connected component in the detected foreground. If the number of pixels in the largest connected component is greater than a preset threshold, then the testing should be continued. This is because the presence of foreground will constitute of a large connected component in the output of the algorithm. If the pixel count of the connected component is lesser than the threshold, there is no moving object in the video thus entering the training period. But in practice, we cannot consider the largest connected component alone. For instance, if a car enters into the camera's field of interest and then parks within the field, the algorithm identifies the moving target as the car and still continues to detect it as the foreground even when the car is parked. Hence, the algorithm will not enter into the training phase. For such applications, in practical cases, after the car has stopped, the algorithm should not further identify the car as foreground. Therefore, in addition to finding the largest connected component we also need to take into account the change in the position of the foreground. This can be done by comparing the location of the detected target in the current frame with a set of previous frames to check if the identified foreground is still moving. In the present work, we move forward with the first solution.

4. PERFORMANCE TUNING AND PARALLELIZATION

This chapter mainly describes the implementation of PracReProCS algorithm in C++, fine tuning of parameters in the code and parallelization of the algorithm using GPUs. The first section discusses about the conversion of code from Matlab to C++ and the advantages of writing the code in C++. The second section describes the fine tuning of parameters. The third section includes profiling of the code. The fourth and fifth part explains the parallelization of SVD and l1 minimization respectively.

4.1 Conversion of code to C++

The PracReProCS algorithm was implemented and published only in Matlab [3]. The code was converted to C++ to provide efficient ways to parallelize the code and for performance improvement. Parallel processing in Matlab is multi-processor based whereas it is multi-thread based in C++ [14]. Therefore, performing many tasks in parallel will give a linear gain upto the number of threads in C++ whereas no significant gain will be observed in Matlab. Also, C++ provides easy use of pointers when passing large amount of memory to functions.

Another advantage of conversion of code to C++ is to make use of OpenCV library as OpenCV has C++ interface. OpenCV is used for the easiness of dealing with image/video processing operations. Further, OpenCV library includes GPU modules that helps in accelerating Matrix operations (multiplication, transpose) and mathematical/bitwise operations (sqrt, min, bitwise_or).

The original algorithm uses Yall1 package [15] for solving the l1 minimization problem in eqn. (2.1). As the type of input parameters for the l1 minimizer was not matching with any of the existing packages in CUDA, the function for l1 minimzation was written separately in C++ by following the same logic in Yall1 package.

4.2 Fine tuning of parameters

During the testing period, some amount of noise was observed when the true foreground was not present. It was observed that two parameters in the algorithm played an important role in the behaviour of noise.

The first parameter was the size of the training dataset. When sufficiently large training set was given, the noise reduced significantly for a slightly moving background. However, increasing the number of samples for training did not remove the noise completely. Also, no improvements were shown when the algorithm was tested on a video that has stationary background.

The second parameter was the threshold that was set after l1 minimization (eqn 2.6). In the original algorithm upon computing l1 minimzation, we get an approximation of the foreground. The threshold was set to remove the sparse signals other than the foreground. This is because of the low threshold value. Even in the testing of parent algorithm some noise was observed when the true foreground was absent. When the threshold was increased to a relatively high value the noise decreased significantly.

4.3 Code profiling

The C++ version of PracReProCS algorithm was tested on the video streamed from the IP camera. After setting up the camera, the initial 60 frames are given for training. In this work, we are training only once after installing the camera. The subsequent incoming frames are given for testing. The computation time for training is 3.6 seconds and that for testing is around 0.7 seconds per frame.

Here, we use clock() function under time.h headerfile to retreive the execution time of a piece of code. The clock() function is called before and after the part of code where we need to find the computation time. Finally, the returned values of the function are subtracted and divided by the number of clock ticks per second to get the computation

time of the code part of interest.

After profiling, the portion of code that consumed most of the time was in SVD. SVD was computed to find the initial basis for the background in training period in eqn. (2.2). The computation time of SVD depends on the size of the training dataset. It took about 3.6 seconds to compute the SVD for 60 training points. However, if we decide to do the training only once after the installation of the camera, the delay in estimating the initial basis need not be taken into consideration.

The next step is to find the time consuming parts in the testing phase. SVD computation in the background updation step is one of the major computationally intensive step in testing. For every α frames we compute the SVD of the projection of the last $\alpha\hat{L}_t s$ frames perpendicular to the previous basis $\hat{P}_{j-1}$. Here, the SVD computation time depends on the value of α as the size of matrix in SVD depends on α . The time for SVD computation for $\alpha = 20$ is approximately 0.2 seconds per frame.

Another part of code which took higher execution time in testing is l1 minimization in eqn. (2.6). for finding an estimate of foreground. The l1 minimizaton took an approximate computation time of 0.5 seconds per frame which also constitutes majority of the time in testing. Further detailed profiling showed that the convergence step in the l1 minimzation took most of the time in l1 minimizer.

4.4 SVD parallelization

The SVD parallelization was done using cuSolver library [16], which is a high-level package based on the cuBLAS[17] and cuSPARSE libraries. In this work, we use cuSolver package in conjunction with cuBLAS library, which is a part of CUDA. The program needs to allocate memory in GPU for the storage of data. Next step is to call a sequence of CUBLAS functions required for the specific application. Then we need to copy back the output from GPU to CPU.

All CUBLAS library function calls return the error status cublasStatus_t. The handle should be initialized by calling cublasCreate() function. The handle is then passed to all the following function calls. Finally, after the usage of the library cublasDestroy() function is called to release the resources associated with the library.

```
cusolverStatus_t = cusolverDnSgesvd (
                        cusolverDnHandle_t handle, signed char jobu, signed char jobvt,
                        int m, int n, float *A, int lda, float *S, float *U,
                        int ldu, float *VT, int ldvt,
                        float *work, int lwork, float *rwork, int *devInfo);
```

Figure 4.1: cusolverDnSgesvd() function

cuSolverDn API provides a subset of dense LAPACK functions. cuSolverDn<t>gesvd() function in the eigen value solver API of cuSolverDn was used to compute SVD. <t> can be either S,D,C or Z which denotes single precision, double precision, complex valued single and double precision datatypes respectively. In this work, we use single precision data and hence use cuSolverSgesvd() function (see Figure 4.1).

The two input parameters jobu and jobvt (Table 4.1) helps in computing all or part of the matrix. But currently, only input 'A' is supported as inputs to both the parameters. Hence, we find the SVD of the full matrix.

The matrix, Mt on which we need to compute the SVD is not square in this work. By default, if the matrix is not square, the full-size square matrices U and Vt will not be calculated (Table 4.2).

Here, we could clearly observe that OpenCV do not find full-sized orthogonal matrices, while cuSolver finds the full-sized matrices.However, we could save some time by copying only the part of the result that we need from GPU to CPU.

Parameter	Meaning
handle	handle to cuSolverDN library
jobu	specifies options for computing all or part of the matrix U = 'A': all m columns of U are returned in U 'S': the first min(m,n) columns of U are returned in U 'O': the first min(m,n) columns of U are overwritten on the array A 'N': no columns of U are computed
jobvt	same function as jobu but applies to vector VT
m	number of rows of matrix A
n	number of columns of matrix A
A	array of dimension lda * n with lda not less than max(1,m)
lda	leading dimension of A
S	singular values of A
U	contains mxm unitary matrix U
ldu	leading dimension of U
VT	nxn unitary matrix VT
ldvt	leading dimension of Vt
work	working space, array of size lwork
lwork	size of work
rwork	unconverged superdiagonal elements of an upper bidiagonal matrix if devInfo > 0
devInfo	devInfo = 0, operation is successful; devInfo = -i, i-th parameter is wrong; devInfo > 0, indicates how many superdiagonals of an intermediate bidiagonal form did not converge to zero

Table 4.1: API of gesvd

MODIFY_A	modifies decomposed matrix to save space and speed up processing
NO_UV	a vector of singular values w are processed; u and vt are empty matrices
FULL_UV	if FULL_UV fag is specified, u and vt will be full-sized orthogonal matrices ; otherwise when matrix is not square, by default algorithm gives u and vt matrices of large size for further A reconstructor

Table 4.2: SVD flags in OpenCV

The SVD computed using cuSolver took 10 seconds to run. The CPU performed SVD computation in 0.25 seconds. One reason could be because of the computation of the

full-sized orthogonal square matrices in SVD in cuSolver. Another reason could be that SVD parallelization is highly dependent on serial factorization iteration. GPU cores have low performance than CPU cores as CPU has higher per core performance. Hence, GPUs would require very large parallelization. Therefore, a better idea would be to write our own SVD algorithm by rethinking about the algebra behind SVD parallelization and avoid computing complete factorization.

4.5 l1 parallelization

l1 took approximately 0.5 seconds per frame to run. Further profiling, we could see that the convergence of l1 minimization consumes 95% most of the time. The convergence took 0.007 seconds per iteration. It was noted that l1 minimization converges in about 70 iterations in most cases, though the maximum number of iterations was 9999.

Most part of the portion of the l1 code used OpenCV GPU module for computation. Further, matrix multiplication needs clBLAS library as OpenCV library version does not support multiplication in gpu. One of the most used matrix multiplication implementation is GEMM in BLAS-3 functions, which contains functions that perform matrix-matrix operations.

After parallelizing the l1 convergence portion in l1 minimization took 0.01 seconds to Careful profiling showed that without tranferring memory between GPU and CPU, 0.004 seconds was taken per iteration. Time taken for 70 iterations in normal case 0.007*70 = 0.49 and that in GPU 0.004*70 = 0.28. Therefore, if the memory transfer was fast, the GPU doubles the performance of the CPU.

5. CONCLUSION & FUTURE WORK

5.1 Conclusion

One of the major challenges in digital media was the moving target detection in the presence of slwoly moving background/noise. PracReProCS algorithm overcome that challenge and showed promising results. This work mainly concentrates on the implementation of PracReProCS algorithm for real time implementations.

Firstly, the original Matlab code was converted to C++ to make it flexible for any kind of parallelzation. After careful profiling of C++ code, it was noted that most of the time was consumed in SVD for training. SVD is computed to find an initial background subspace. However, we do not need to consider the training time as we only train the background once after installing the camera.

Part of code in testing	Execution time
Total	0.7 seconds
SVD	0.2 seconds
l1	0.5 seconds

Table 5.1: Distribution of computational time in testing

Next we profiled the testing part of the code. It was observed that SVD computation in background updation step and the l1 minimizer took most of the time in testing.

Parallelization of SVD in CPU is better than GPU due to computation of full sized orthogonal matrices as opposed to OpenCV SVD computation. Also, SVD depends on serial factorization iteration and requires large parallelization. GPU core have low performance than CPU core. Hence, parallelization in GPU will be more time consuming. Further,

GPU/CPU	Execution time per iteration	Total execution time
CPU	0.007 seconds	0.49
GPU (without memory tranfer)	0.004 seconds	0.28
GPU (with memory transfer)	0.008 seconds	0.56

Table 5.2: Distribution of computational time for l1 minimizer

we parallelized convergence part of l1 minimizer. It was observed that the computation speed was twice faster in GPU than CPU. But considering the overhead in memory for gpu transfer, CPU performs better than CPU.

5.2 Future Work

The frame rate of video surveillance camera that we used was 15fps, i.e, in every 0.06 seconds a frame is captured and passed as input to the algorithm. Hence, to implement the algorithm in real time we need to really careful in optimizing the possible steps in the code. Though the computation of l1 minimization was computationally faster in GPU, due to milliseconds delay in the tranfer of data between GPU and CPU, CPU outperformed GPU. Hence, for such real-time applications, it is highly important to consider the overhead in memory for GPU data transfer.

A better architecture would be to model the algorithm such that GPUs does most of the work. The video is streamed and the training is done in CPU. After finding the initial basis for background subspace, the basis vector is passed onto GPU. Further incoming frames are passed to GPU and the GPU does all the remaining steps including l1 minimization. When the background space updation is reached, the results of testing is passed to CPU. This is because background space updation requires SVD computatuion and is faster in CPU.

Further time can be reduced if we make the CPU and GPU run at the same time. Suppose we gave the first tesing frame to CPU. If the second frame arrives before the

whole computation of the algorithm, we can take the second frame and start processing in CPU though the first computation is not over.

Also, we could implement parallelization techinique in CPU like OpenMP, MPI since the board has four cores. This will provide more room for parallelization.

REFERENCES

[1] R. G. Baraniuk, T. Goldstein, A. C. Sankaranarayanan, C. Studer, A. Veeraraghavan, and M. B. Wakin, "Compressive video sensing: algorithms, architectures, and applications," *IEEE Signal Processing Magazine*, vol. 34, no. 1, pp. 52–66, 2017.

[2] H. Guo, C. Qiu, and N. Vaswani, "An online algorithm for separating sparse and low-dimensional signal sequences from their sum," *IEEE Transactions on Signal Processing*, vol. 62, no. 16, pp. 4284–4297, 2014.

[3] http://www.ece.iastate.edu/ hanguo/PracReProCS.html#Code_.

[4] E. Lusk and A. Chan, "Early experiments with the openmp/mpi hybrid programming model," *Lecture Notes in Computer Science*, vol. 5004, p. 36, 2008.

[5] C. Harris, K. Haines, and L. Staveley-Smith, "Gpu accelerated radio astronomy signal convolution," *Experimental Astronomy*, vol. 22, no. 1-2, pp. 129–141, 2008.

[6] J. Kruger and R. Westermann, "Acceleration techniques for gpu-based volume rendering," in *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, p. 38, IEEE Computer Society, 2003.

[7] C. Nvidia, "Compute unified device architecture programming guide," 2007.

[8] R. Laganière, *OpenCV 3 Computer Vision Application Programming Cookbook*. Packt Publishing Ltd, 2017.

[9] J. Moreno, G. Ortega, E. Filatovas, J. Martínez, and E. M. Garzón, "Using low-power platforms for evolutionary multi-objective optimization algorithms," *The Journal of Supercomputing*, vol. 73, no. 1, pp. 302–315, 2017.

[10] J. Feeley, *NVIDIA: A Pinmux Configuration Tool for Tegra*. PhD thesis, Worcester Polytechnic Institute, 2017.

[11] A. J. Lipton, H. Fujiyoshi, and R. S. Patil, "Moving target classification and tracking from real-time video," in *Applications of Computer Vision, 1998. WACV'98. Proceedings., Fourth IEEE Workshop on*, pp. 8–14, IEEE, 1998.

[12] C. Zhan, X. Duan, S. Xu, Z. Song, and M. Luo, "An improved moving object detection algorithm based on frame difference and edge detection," in *Image and Graphics, 2007. ICIG 2007. Fourth International Conference on*, pp. 519–523, IEEE, 2007.

[13] http://developer.download.nvidia.com/embedded/jetson/TK1/docs/3_HWDesignDev/JTK1_DevKit_Specification.pdf.

[14] https://stackoverflow.com/questions/20513071/performance-tradeoff-when-is-matlab-better-slower-than-c-c.

[15] Y. Zhang, "User′s guide for yall1: Your algorithms for l1 optimization," *Technique report*, pp. 09–17, 2009.

[16] http://docs.nvidia.com/cuda/cusolver/index.html.

[17] http://docs.nvidia.com/cuda/cublas/index.html.